HIGH FREQUENCY DATA MANAGEMENT (HFDM)

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims the benefit under 35 U.S.C. Section 119(e) of the following co-pending and commonly-assigned U.S. provisional patent application(s), which is/are incorporated by reference herein:

[0002] Provisional Application Ser. No. 62/907,173, filed on Sep. 27, 2019, with inventor(s) Dov Amihod, Thiago da Costa, Arno Zinke, Sebastian Medan, Farzad Towhidi, and Roland Artur Ruiters-Christou, entitled "High Frequency Data Model (HFDM)," attorneys' docket number 30566. 0584USP1.

BACKGROUND OF THE INVENTION

1. Field of the Invention

[0003] The present invention relates generally to collaborative design, and in particular, to a method, apparatus, system, and article of manufacture for a high frequency data management system that solves latency and user workflow issues.

2. Description of the Related Art

[0004] (Note: This application references a number of different publications as indicated throughout the specification by reference numbers enclosed in brackets, e.g., [x]. A list of these different publications ordered according to these reference numbers can be found below in the section entitled "References." Each of these publications is incorporated by reference herein.)

[0005] Collaborative design is becoming a bigger area and accepted workflow. The number of users on any one project can expand very quickly and result in changes and forks, that can slow performance and complicate the workflow. Prior art collaborative systems fail to store data in an efficient, fast, and accessible manner that can be utilized by multiple users in a collaborative system. More specifically, it is desirable to store data in a manner that is performed in an automated manner, that is efficient and processed quickly both across a network and locally, with simultaneous access to collaborating users, where version history and reversion are available. To better understand these problems, a description of prior art data management systems and their limitations may be useful.

[0006] Various types of applications (e.g., content creation applications) impose demanding requirements on their data infrastructure. Data grows larger over the lifetime of assets, projects and companies. There are intense sharing/collaborative challenges, with very high-expectations with regards to low latency, real-time feedback, data-availability and durability. Further, many people simply don't know they have a problem with keeping history for the content they generate. Specifically, many types of applications require the history of data to be preserved—for example financial applications, document management applications or content creation applications. The most challenging applications to build are modern content creation applications in the web. Users expect their work to be always saved, and that it can be reverted to earlier versions. Users may go offline for periods of time and need to reconcile multiple versions of documents—by multiple collaborators. They also expect real-time collaborative editing, like in GOOGLE DOCS, and an interactive application that quickly reacts to changes.

[0007] Such a requirement of historical data imposes high demands on the data infrastructure of such applications. For example, a distributed system may have several distributed nodes that have to be synchronized—sometimes in real time, sometimes after long periods of time. Such a system must cope with a high rate of data changes (at the speed of key presses or mouse movements), and needs to support rich data-models such as the ones needed for web-page editors, CAD applications or document editors. In addition, such systems may have to cope with large amounts of data and provide high availability and durability.

[0008] To realize such applications, it would be useful to have a system that allows the following functionality to be realized at the same time: a) complete change history, b) client-level state while globally eventually consistent, c) branching & merging of change histories, d) random access to all data, e) efficient access to the state of all data at any point in history.

[0009] Many products that create content have their data-backends designed without knowledge of these problems/design requirements. Most products approach data management for content as a virtual file system. This approach kicks off a chain of future problems and limitations with regards to management of user generated content in the presence of concurrent editing. Please note that this includes not only multi-user collaboration use cases but also scenarios where services produce or consume data, particularly in the context of a distributed computer.

[0010] There are several systems that implement a subset of the requirements defined above. For example, AWS APPSYNC [4], SHARE.JS [5] AND GOOGLE CLOUD FIRESTORE [6] implement a distributed system with a local state and conflict resolution logic to handle competing writes, but they don't provide complete change history with branching. On the other hand, there are systems that provide full version history with branching on complex data-structures, for example for SQL databases [7] or XML documents [8], but which are not implemented as distributed systems.

[0011] Prior art systems may provide a class of distributed data storage systems that make history their primary concern (GIT [1] and BLOCKCHAIN). However, such change-based systems operate on the basis of storing data-changes.

[0012] One may also note that the above described requirements seem to be interdependent. Change-based data systems seem to suffer from the same set of challenges—for example random data access is difficult while encoding history—which warrants studying these systems as a class of systems. Achieving one requirement does not guarantee one can achieve the next requirement.

[0013] More specifically, GIT [1] provides a distributed revision control system, that uses local states on the clients and uses branching and merging/rebasing to achieve eventual consistency. However, GIT's main use case is versioning of source code. It therefore manages a folder tree, instead of a more fine-grained document model of embodiments of the present invention. Further, GIT focuses on the low frequency use cases, in which usually only a few synchronizations per day happen, instead of constantly keeping multiple nodes in sync.

[0014] TARDiS [2] also follows an approach where clients have a local state and branches. Conflicts are resolved via